RE-Introduction to One-Sided communication

PGAS Languages: Co-arrays in Fortran 2008 UPC

Nathan Wichmann
wichmann@cray.com

CRAY
THE SUPERCOMPUTER COMPANY

Outline

- What is one-sided communication?
- How do I do this?
- Why would I want to?
- Examples and success stories.

PGAS programming

- **P**artitioned **G**lobal **A**ddress **S**pace
- Language level parallelism as opposed to library calls
  - Extension to C – Unified Parallel C (UPC)
  - New feature call Co-array in Fortran 2008 (CAF)
- Single-sided communication as opposed to two-sided MPI comms
- Explicit synchronization required – this is (mostly) implicit in MPI
- Gives compiler lots of freedom for optimization
- Many algorithms are very naturally expressed using one-sided language level parallelism
  - Handing off work/data to another pe
  - Halo exchanges
  - Mesh manipulation and movement

PGAS and Cray

- Cray have been supporting CAF and UPC since the beginning
  - Support on the T3E, Cray X1, X2
- Full PGAS support on the Cray XT
  - Cray Compiling Environment 7.0 – Dec 08
  - Full UPC 1.2 specification
  - Full CAF support – CAF proposed for the Fortran 2008 standard
  - Hybrid MPI/PGAS codes supported – very important!
- Fully integrated with the Cray software stack
  - Same compiler drivers, job launch tools, libraries
  - Integrated with Craypat – Cray performance tools

Special features of Baker relating to CAF/UPC

- On X1 and X2, the custom processor directly emits addresses for any memory location in the machine. Loads/ stores can be done to any global address in the system

- On Baker the Gemini NIC used to ‘extend’ address space of Opteron references to access memory on remote nodes
  - Fortran or C compilers recognize CAF/UPC references and generates appropriate messages to Gemini to load from or store to remote memory
  - Users can stride on local offsets or across processor space with any stride, including Gather/Scatter

Fortran 2008 - Parallel programming

- Fortran 2008 is a natively parallel language

- SPMD programming model
- Simple syntax for one-sided communication
- Image synchronization
- Coordinated program termination

Fortran 2008 - programming model

Executable is replicated across processors (MPI-like)

Each instance is called an “IMAGE”

Each image has its own data objects

Each image executes asynchronously except when syncs are indicated

Co-array Fortran: Basics and terminology

- Any time a co-array appears without [ ]'s, the reference is to the data on the local image
- The number inside the [ ]'s can reference any image in the job, including myself
- If a reference with [ ]'s appears to the right of the =, it is often called a "get"
- If a reference with [ ]'s appears to the right of the =, it is often called a "put"

What does this do? a(:)[ri] = b(:)

The statement copies, or "puts" a local "b" into the "a" of image "ri"

CAF array syntax

- Declaration

```
real(8), ALLOCATABLE :: rcvbuf(:,:)[:]
```

- Dynamic Allocation

```
! Allocate m*n elements on each processor
  ALLOCATE( rcvbuf(m,n)[*] )
```

- Reference

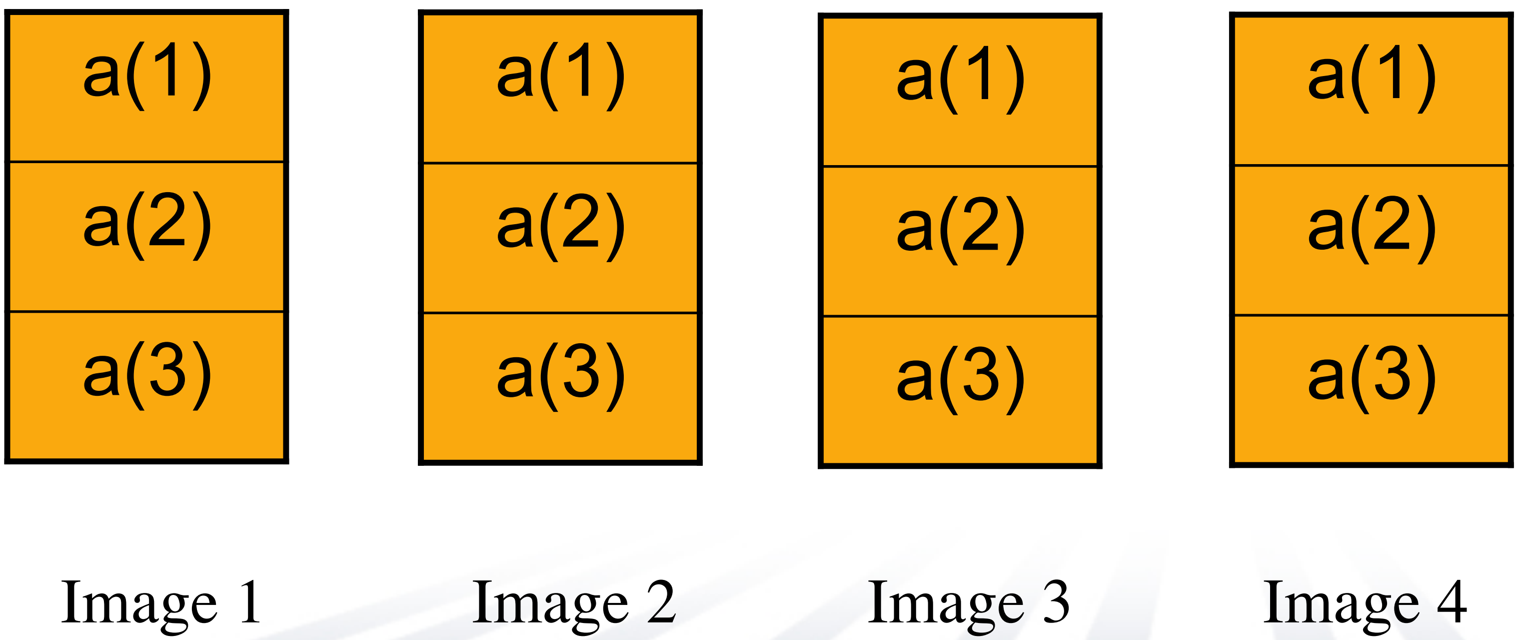
```
! Reference element (i,j) on processor k
  rcvbuf(i,j)[k]
```

- PE control

```
this_image(), num_images()
```

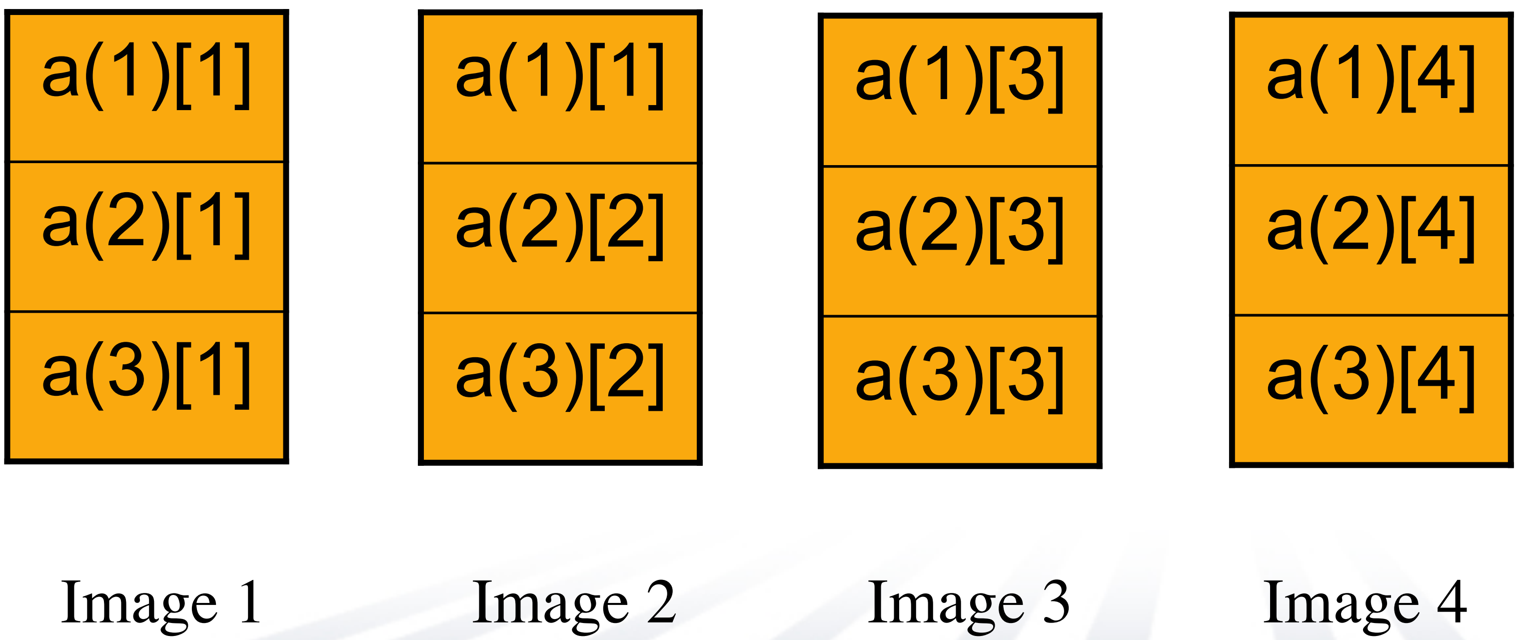
Array Example

Real(8) a(3)



Co-Array Example

Real(8) a(3)[*]



Fortran 2008 – Basic “Put”

```
real :: s(100)[*]
real,allocatable :: a(:)[:]     !  S and A are “co-arrays”

allocate (a(100)[*])
a = 10.
s = 11.
mype = this_image()

if (mype == 1)  a(:)[1] = s(:)[2]
```

Fortran 2008 - synchronization

Explicit statements:

sync all
sync images (images)
sync memory
critical / end critical
lock / unlock

Implicit synchronization:

allocation of a co-array
deallocation of a co-array (either explicit or implicit)

RYO synchronization:

atomic_ref / atomic_def

Halo Exchange: MPI

```
doubleprecision ai(ip,ihp,6)
...
call mpi_isend ( ai(1,1,1),   ihp*ip,   mpi_real, imgi(myp+1), &
                    9905, mpi_comm_world, mpireq(1), mpierr );
call mpi_isend ( ai(1,1,2), 2*ihp*ip,   mpi_real, imgi(myp-1), &
                    9906, mpi_comm_world, mpireq(2), mpierr );
call mpi_irecv ( ai(1,1,4),   ihp*ip,   mpi_real, imgi(myp-1), &
                    9905, mpi_comm_world, mpireq(3), mpierr );
call mpi_irecv ( ai(1,1,5), 2*ihp*ip,   mpi_real, imgi(myp+1), &
                    9906, mpi_comm_world, mpireq(4), mpierr );
call mpi_waitall ( 4, mpireq, mpistat )
```

Each PE must make a call to MPI to do BOTH the send and the receive. Both PE's must know the communication will happen and perform the message passing "at the right time".

Halo Exchange: Co-array Fortran

```
Real(8) ai(ip,ihp,6)[*]
....
ai(:,:,4:4) = ai(:,:,1:1)[img(myp-1)]
ai(:,:,5:6) = ai(:,:,2:3)[img(myp+1)]
call sync_all()
```

Simple, transparent syntax. The other PE does not need to directly participate

One only needs to know there are not race conditions on the data.

What if one cannot change variables?

- Subroutine dummy argument

```
SUBROUTINE fft_transpose( zstick, . . . )
 COMPLEX :: zstick( * )
```

- Define derived type

```
TARGET zstick

 TYPE CAFP
   COMPLEX, DIMENSION(:), POINTER :: p
 END TYPE CAFP
 TYPE (CAFP) pzstick[*]
```

What if cannot change (cont'd)

- Set pointer

```
pzstick%p => zstick(1:n)
 call sync_all()
```

- Reference zstick(i) on proc k

```
pzstick[k]%p(i)
```

Fortran 2008 - pointer components

```
subroutine sort(t,n)
integer :: n
real,target :: t(n)

type yy
   real,pointer :: p(:)
end type yy
type(yy) :: image[*]

image%p => t          !  Point at the remote data
sync all              !  Sync to make sure everyone has completed the pointing
x = image[1]%p(1)  ! Reference the remote data
…
```

CAF works very well with OpenMP

```
!$omp parallel do …
do j=1,m
 do i=1,n
  target(i,j)[target_proc] = source(i,j)
enddo;enddo
```

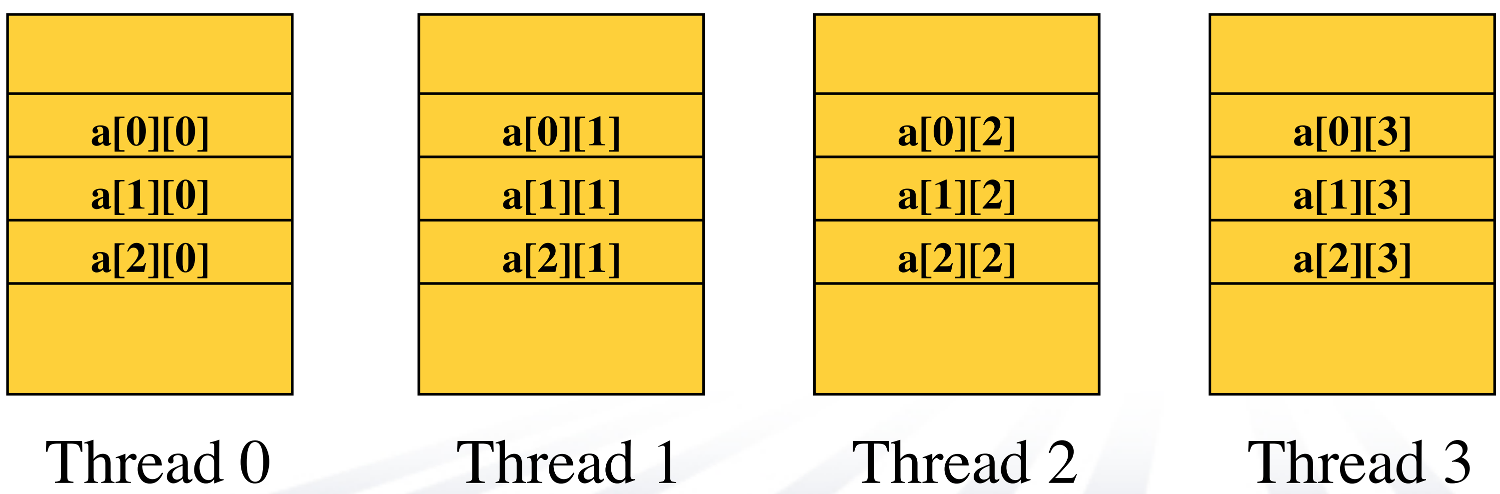
Uses all OMP threads to “put” data into the target location on the target processor.

What is UPC?

- Unified Parallel C
- Syntactic extension to Standard C
- Designed by IDA CCS, UC-Berkeley, LANL
- SPMD plus HPF-like data and work distribution features

UPC Examples

shared double a[3][THREADS];



Halo Exchange: Cray UPC Subset

```
shared double ai[6][ihp][ip][THREADS];
...
for ( j=0; j<ihp; j++ ){
    for ( i=0; i<ip; i++ ){
        ai[3][j][i][MYTHREAD] = ai[0][j][i][thd[myp-1]]
        ai[4][j][i][MYTHREAD] = ai[1][j][i][thd[myp+1]];
        ai[5][j][i][MYTHREAD] = ai[2][j][i][thd[myp+1]];
    }
}
upc_barrier();
```

Advantages very similar to CAF.

Advantages of One-Sided communication

- Easier to program (See Halo Exchange and Random Access)
  - Don’t have to write both send and receive
  - More transparent what is going on
- Enables new algorithms (See Dynamic Mesh CFD from AHPCRC)
  - Only one PE needs to do the communication. Data can be retrieved and modified without coordination with PE holding the data
  - More freedom in when and where the communication is done.
- Reduced communication time overhead, hence better scalability (for supporting hardware)
- Can spread out communications / Easier to Mix computation and communication
  - Collectives like MPI_ALLTOALL concentrate communication
  - Perhaps better to get and put data as you need it.

Good Practices for UPC/CAF/shmem, etc.

- Use 4/8 byte sized shared variables
- Code to overlap communication with computation
  - Hardware is designed to support this
  - Reduces communication hot spots
- Try to “PUT” data instead of “GET” data
- Avoid use of *strict* shared types when possible (increases memory synchronization requirements)
- Try to avoid generalized scatter/gather in inner loops (unless overlapped with computation)

PGAS in practice:

Examples of how PGAS was, can, and will be used

Smoothing out Communication

- MPI collectives can often focus all communication into a specific time and place
- Can use CAF/UPC to spread that communication out and overlap with computation
- Simplified MVH3:

```
do i=1,many
  call sweepx        ! All compute, no communication
  call mpi_alltoall  ! Intense communication
  call sweepy        ! All compute, no communication
  call mpi_alltoall  ! Intense communication
  call sweepz        ! All compute, no communication
  call mpi_alltoall  ! Intense communication
enddo
```

Smoothing out Communication

- Put data to next location as you compute it

```
do i=1,many
  call sweepx_put_to_y        ! Compute, with communication spread out
  call sweepx_put_to_y        ! Compute, with communication spread out
  call sweepx_put_to_y        ! Compute, with communication spread out
enddo
```

GTC: Online/asynchronies diagnostic processing*

- GTC requires diagnostic to be run at the same time as simulation
  - Too much data to store for post-processing
  - Does not need to prevent the simulation from proceeding
- One group of processors can process data while main simulation process
  - How does one coordinate the data transfer?

* Thanks to Scott Klasky of ORNL for this idea

GTC: Online/asynchronies diagnostic processing

- Simulation group

```
do i=1,timestep

  call main_computation

  call check_if_diagnostic_buffer_read

  call put_data_to_diag

  call inform_diag_data_is_ready

enddo
```

- Diagnostic Group

```
do i=1,timestep

  call set_buffer_ready_flag

  call wait_for_data

  call perform_diagnostics

  call store_diagnostic_results

enddo
```

Simple double buffer can help reduce synchronization cost to close to zero

UPC Random Access: Designed for Speed

- This version of UPC Random Access was originally written in Spring 2004
- Written to maximize speed
- Had to work inside of the HPCC benchmark
- Had to run well on any number of CPUs
- Also happens to be a very productive way of writing the Global RA.

Productivity: Fewer lines of code

UPC VERSION

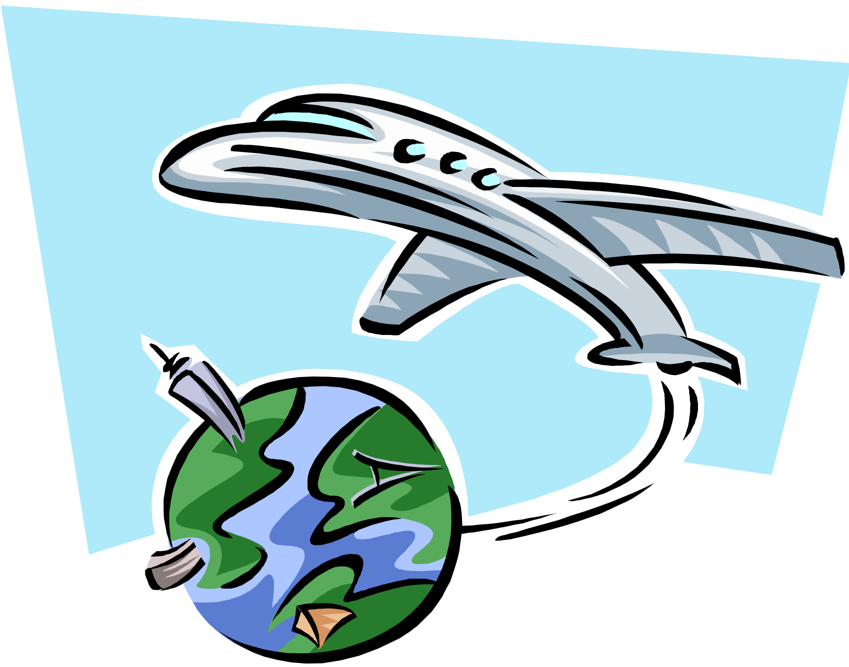
```
#pragma _CRI concurrent
for (j=0; j<STRIPSIZE; j++)
 for (i=0; i<SendCnt/STRIPSIZE; i++) {
  VRan[j] = (VRan[j] << 1) ^ ((s64Int) VRan[j]< ZERO64B ?
     POLY : ZERO64B);
  GlobalOffset = VRan[j] & (TableSize - 1);
  if (PowerofTwo) LocalOffset=GlobalOffset>>logNumProcs ;
  else            LocalOffset=(double)GlobalOffset/
     (double)THREADS;
    WhichPe=GlobalOffset-LocalOffset*THREADS;
    Table[LocalOffset][WhichPe] ^= VRan[j] ;
 }
}
```

BASE VERSION

```
NumRecvs = (NumProcs > 4) ?(Mmin(4,MAX_RECV)) : 1;
  for (j = 0; j < NumRecvs; j++)
    MPI_Irecv(&LocalRecvBuffer[j*LOCAL_BUFFER_SIZE],
    localBufferSize,INT64_DT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD,&inreq[j]);
while (i < SendCnt) {
do {
MPI_Testany(NumRecvs, inreq, &index, &have_done,
    &status);
if (have_done) {
 if (status.MPI_TAG == UPDATE_TAG) {
   MPI_Get_count(&status, INT64_DT, &recvUpdates);
bufferBase = index*LOCAL_BUFFER_SIZE;
for (j=0; j < recvUpdates; j ++) {
 inmsg = LocalRecvBuffer[bufferBase+j];
 LocalOffset = (inmsg & (TableSize - 1)) -
    GlobalStartMyProc;
 HPCC_Table[LocalOffset] ^= inmsg;
 }
 } else if (status.MPI_TAG == FINISHED_TAG) {
    NumberReceiving--;
 } else {
    abort();
 }
```

Productivity : Fewer lines of code

UPC VERSION

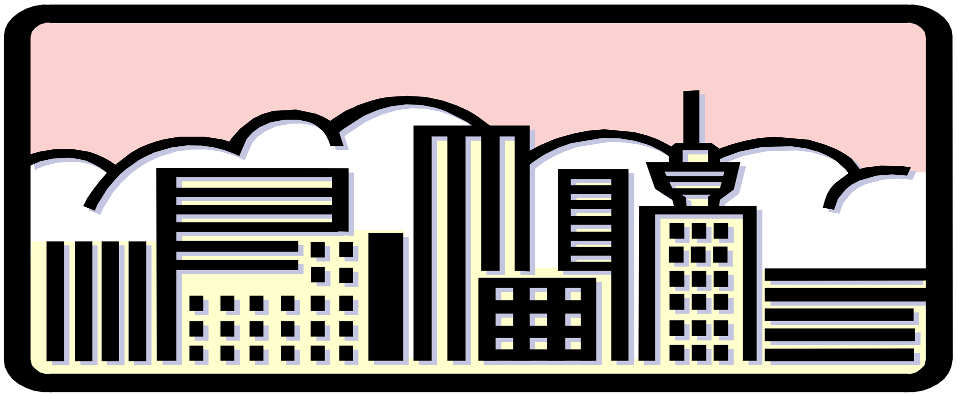


BASE VERSION

```
MPI_Irecv(&LocalRecvBuffer[index*LOCAL_BUFFER_SIZE],
    localBufferSize,INT64_DT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD,&inreq[index]);
}
} while (have_done && NumberReceiving > 0);
  if (pendingUpdates < maxPendingUpdates) {
    Ran = (Ran << 1) ^ ((s64Int) Ran <   ZERO64B ?
    POLY : ZERO64B);
    GlobalOffset = Ran & (TableSize-1);
    if ( GlobalOffset < Top)
      WhichPe = ( GlobalOffset / (MinLocalTableSize +
    1) );
   else
    WhichPe = ( (GlobalOffset - Remainder) /
    MinLocalTableSize );
   if (WhichPe == MyProc) {
    LocalOffset = (Ran & (TableSize - 1)) -
    GlobalStartMyProc;
    HPCC_Table[LocalOffset] ^= Ran;
   }
   else {
    HPCC_InsertUpdate(Ran, WhichPe, Buckets);
        pendingUpdates++;
   }
   i++;
 }
 else {
```

Productivity : Fewer lines of code

UPC VERSION



BASE VERSION

```
MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
   if (have_done) {
    outreq = MPI_REQUEST_NULL;
    pe = HPCC_GetUpdates(Buckets, LocalSendBuffer,
    localBufferSize, &peUpdates);
    MPI_Isend(&LocalSendBuffer, peUpdates, INT64_DT,
    (int)pe, UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
   }}}
while (pendingUpdates > 0) {
do {
MPI_Testany(NumRecvs, inreq, &index, &have_done,
    &status);
if (have_done) {
  if (status.MPI_TAG == UPDATE_TAG) {
   MPI_Get_count(&status, INT64_DT, &recvUpdates);
   bufferBase = index*LOCAL_BUFFER_SIZE;
  for (j=0; j < recvUpdates; j ++) {
   inmsg = LocalRecvBuffer[bufferBase+j];
   LocalOffset = (inmsg & (TableSize - 1)) -
    GlobalStartMyProc;
   HPCC_Table[LocalOffset] ^= inmsg;
   }
} else if (status.MPI_TAG == FINISHED_TAG) {
  NumberReceiving--;
```

Productivity : Fewer lines of code

UPC VERSION

BASE VERSION

```
} else {
  abort();}
MPI_Irecv(&LocalRecvBuffer[index*LOCAL_BUFFER_SIZE],
    localBufferSize,INT64_DT, MPI_ANY_SOURCE, MPI_ANY_TAG,
    MPI_COMM_WORLD,&inreq[index]);
}} while (have_done && NumberReceiving > 0);
    MPI_Test(&outreq, &have_done, MPI_STATUS_IGNORE);
    if (have_done) {
      outreq = MPI_REQUEST_NULL;
      pe = HPCC_GetUpdates(Buckets,     LocalSendBuffer,
    localBufferSize, &peUpdates);
    MPI_Isend(&LocalSendBuffer, peUpdates, INT64_DT,
    (int)pe, UPDATE_TAG, MPI_COMM_WORLD, &outreq);
    pendingUpdates -= peUpdates;
    } }
for (proc_count = 0 ; proc_count < NumProcs ; +
    +proc_count) {
  if (proc_count == MyProc) { finish_req[MyProc] =
    MPI_REQUEST_NULL; continue; }
  MPI_Isend(&Ran, 1, INT64_DT, proc_count,
    FINISHED_TAG,MPI_COMM_WORLD, finish_req + proc_count);
  }
while (NumberReceiving > 0) {
```

Productivity : Fewer lines of code

UPC VERSION

BASE VERSION

```
MPI_Waitany(NumRecvs, inreq, &index, &status);
if (status.MPI_TAG == UPDATE_TAG) {
  MPI_Get_count(&status, INT64_DT, &recvUpdates);
  bufferBase = index * LOCAL_BUFFER_SIZE;
for (j=0; j < recvUpdates; j ++) {
  inmsg = LocalRecvBuffer[bufferBase+j];
  LocalOffset = (inmsg & (TableSize - 1)) -
    GlobalStartMyProc;
  HPCC_Table[LocalOffset] ^= inmsg;
  }
  } else if (status.MPI_TAG == FINISHED_TAG){
     NumberReceiving--;
    } else {
      abort(); }
MPI_Irecv(&LocalRecvBuffer[index*LOCAL_BUFFER_SIZE],
    localBufferSize,INT64_DT, MPI_ANY_SOURCE,
    MPI_ANY_TAG, MPI_COMM_WORLD, &inreq[index]);
}
MPI_Waitall( NumProcs, finish_req, finish_statuses);
HPCC_FreeBuckets(Buckets, NumProcs);
for (j = 0; j < NumRecvs; j++) {
    MPI_Cancel(&inreq[j]);
    MPI_Wait(&inreq[j], &ignoredStatus);
  }
```

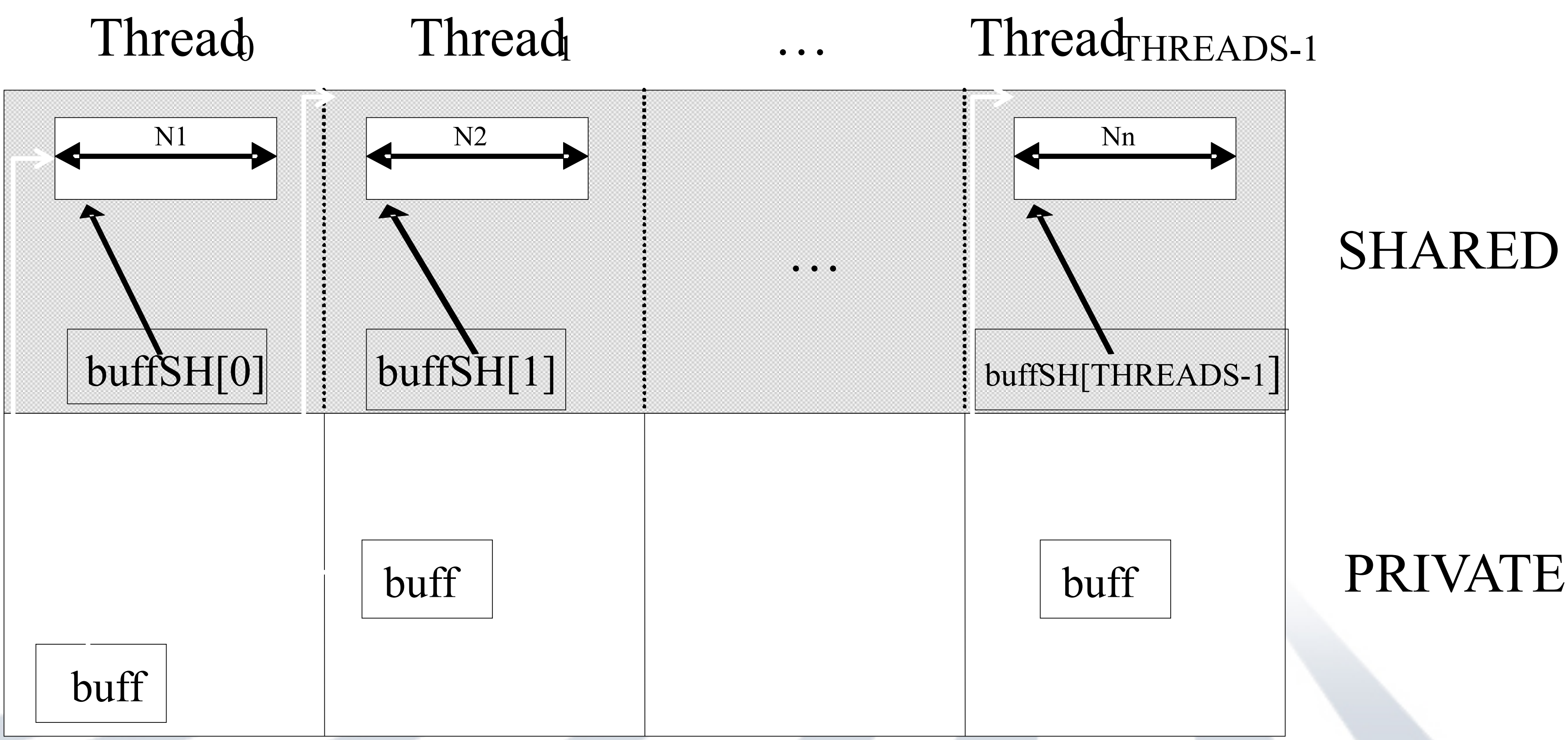
BenchC

- MPI mode
  - All communication and timing done using MPI calls
- UPC mode
  - Problem I/O, setup and initialization done using MPI
  - Core communication (gathers, scatters, collective operations), timing done using UPC

- Communication method (MPI vs UPC) is selected at compile-time
- UPC was incrementally added to the code, one routine at a time.

Data Layout

- Very lightweight modification to the code – little disruption
- Create shared buffers for boundary exchanges
  - shared double *shared buffSH[THREADS]
  - buffSH[MYTHREAD] = (shared double *shared)upc_alloc(nec*sizeof(Element)) – for remote access of shared data
  - buff = (double *) buffSH[MYTHREAD] – for local access of shared data



BenchC UPC Communication Patterns

- Gathers/Scatters

  …initial ordering of data…

  ```
  ip = ep[i];
  for (j = 0; j < num; j++)
    bg[iloc1 + j] = buffSH[ip][iloc2 + j]  /*collect data from across THREADS*/
  ```

- Broadcasts/Reductions

  ```
  global_normSH[MYTHREAD] = loc_val;
  if (MYTHREAD == 0){
     for (ip = 1; ip < THREADS; ip++) loc_val +=global_normSH[ip];
     for (ip = 0; ip < THREADS; ip++) global_normSH[ip] = loc_val;
  }
  ```

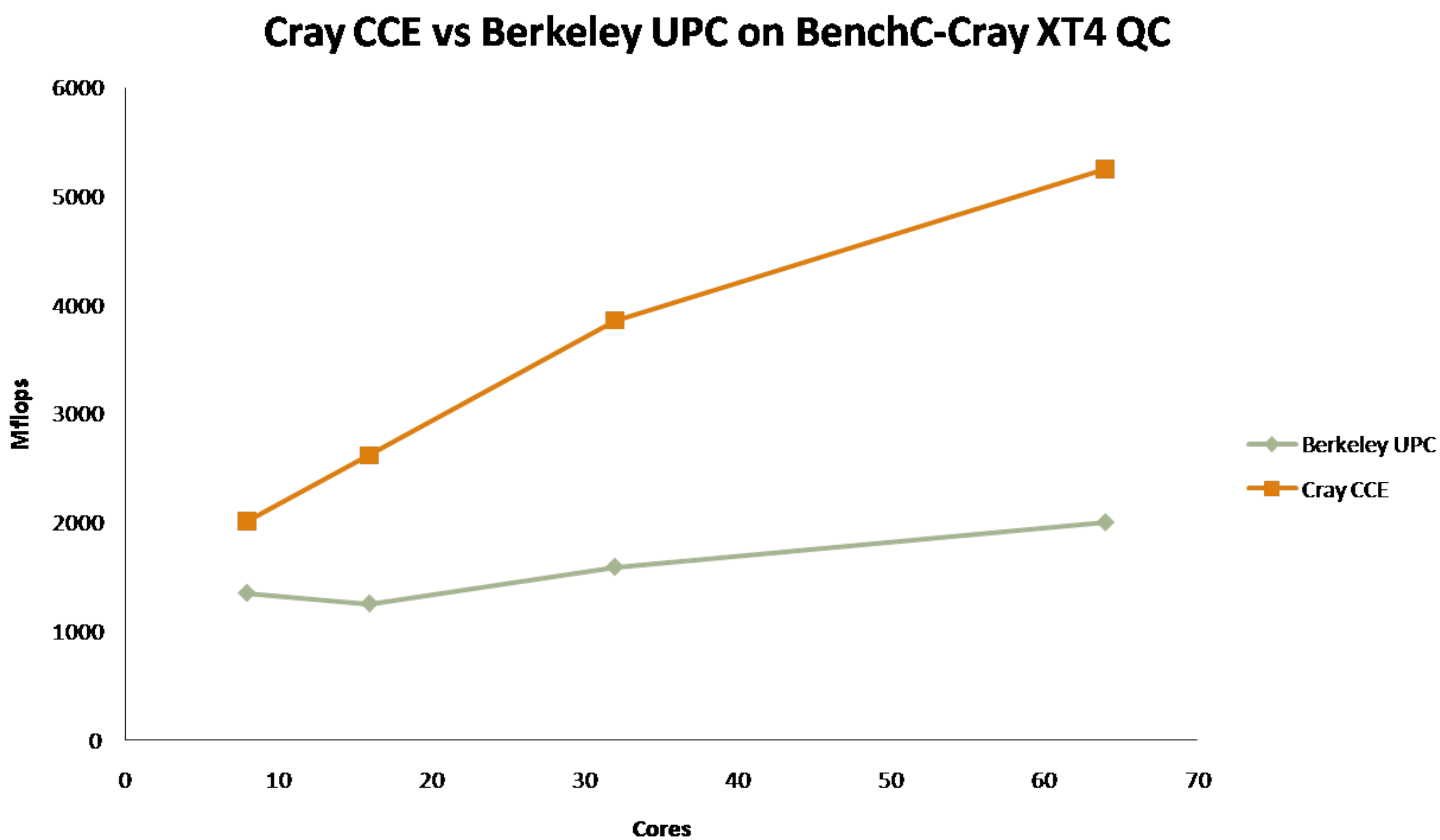
- Code just compiles normally
  - cc –h upc –c –h list=ms my_upc_broadcast.c

UPC BenchC code example-Cray XT5m

```
664.  1-------<    for (i = 0; i < epnum; i++){
665.  1               iloc1 = eploc [i]*len;
666.  1               iloc2 = eploc2[i]*len;
667.  1               num = (eploc[i+1] - eploc[i+0])*len;
668.  1               btSend += num*8;   /* sizeof(double) */
669.  1               ip = ep[i];
670.  1
671.  1          #pragma ivdep
672.  1 r----<>        for (j = 0; j < num; j++) buffSH[ip][iloc2+j] =
                                                   bg[iloc1+j];
```

CC-6005 CC: SCALAR File = ncommsetup.c, Line = 672
A loop was unrolled 8 times.

CC-6325 CC: VECTOR File = ncommsetup.c, Line = 672
Although A loop was marked with an IVDEP directive, it cannot be vectorized because it contains one or more operations that have no vector form.



Cray CCE vs Berkeley UPC on BenchC-Cray XT4 QC
6000
5000
4000
3000
2000
1000
0
Mflops
0
10
20
30
40
50
60
70
Cores
Berkeley UPC
Cray CCE

Dynamic-Mesh Generation Xflow from AHPCRC

- Tightly Couple automatic mesh generation technology within parallel flow solvers
  - Mesh generation never stop and runs in-conjunction with the flow solver
  - Mech continuously changes due to changes in geometry.or other conditions

Source: PGAS presentation of Andrew Johnson of AHPCRC

XFlow

- Complex CFD applications have moving components and/or changing domain shapes
  - Rotational geometries and/or flapping wings
  - Most fluid-structure interaction applications
  - Engines, turbines, pumps, etc.
  - Fluid-particle flows and free-surface flow
- Many methods have been developed to solve these types of applications
  - None are ideal and all have limitations
- MPI approach to “Dynamic-Mesh CFD” was unsuccessful in the past due to algorithm complexity (~1997 time frame)
- Ultimate goal of “Dynamic-Mesh CFD”
  - Mesh should be “dependent” on the solution by continuously changing

XFlow

- Fully integrate automatic mesh generation within the parallel flow solver
  - Mesh generation never stops and runs in-conjunction with the flow solver
    - Element connectivity changes as required to maintain a “Delaunay” mesh
    - New nodes added as required to match user-specified refinement values
    - Existing nodes deleted when not needed
  - Mesh continuously changes due to changes in geometry and/or the solution
    - Mesh size can grow or shrink at each time step
- Very complicated method
  - Parallelism (UPC), vectorization, dynamic data structures, solvers (mesh moving and fluid flow), general CFD accuracy, scalability, CAD links, etc.
  - Will take time to fully evaluate

XFlow

- Mesh is distributed amongst all processors (fairly uniform loading)
- Developed a fast Parallel Recursive Center Bi-section mesh partitioner
- Each processor maintains and controls its own piece of the mesh
  - Each processor has a list of nodes, faces, and elements
  - Each list consists of an array of C structures (Node, Face, or Element arrays)
  - These arrays are defined “shared”
    - Adds a “processor-dimension” to each array

elementsSH[proc][local-index].n[0-through-4]

elementsSH[proc][local-index].np[0-hrough-4]

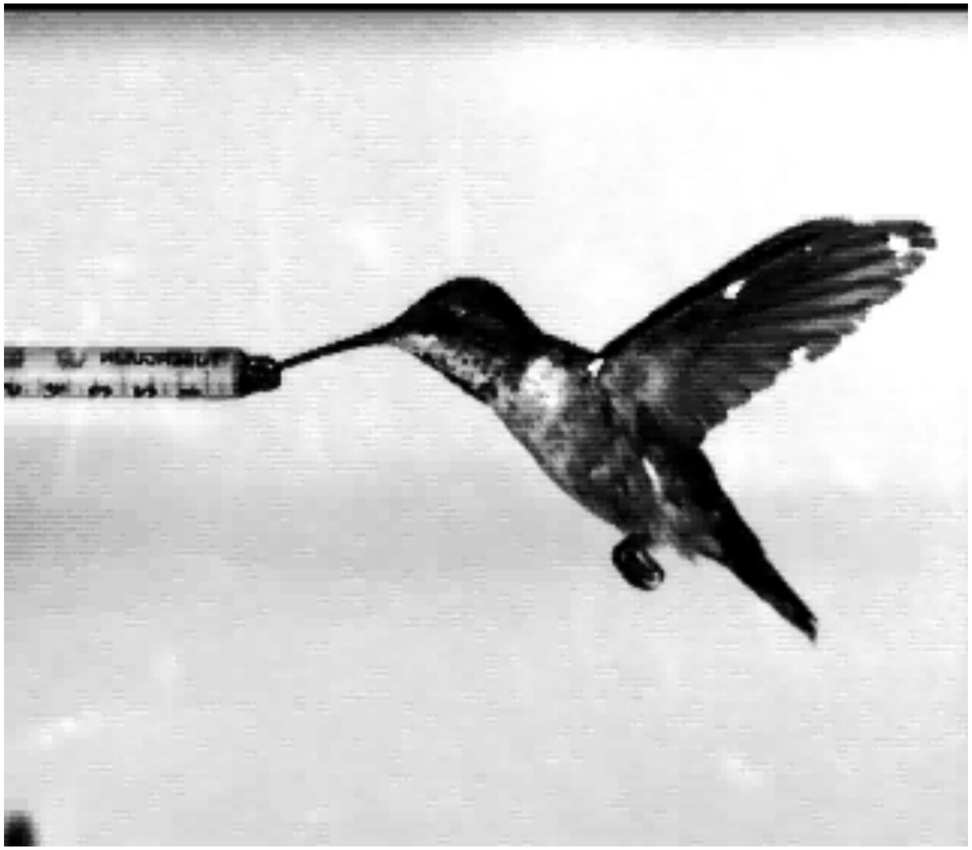
elements[local-index].c[X]

elements[local-index].det

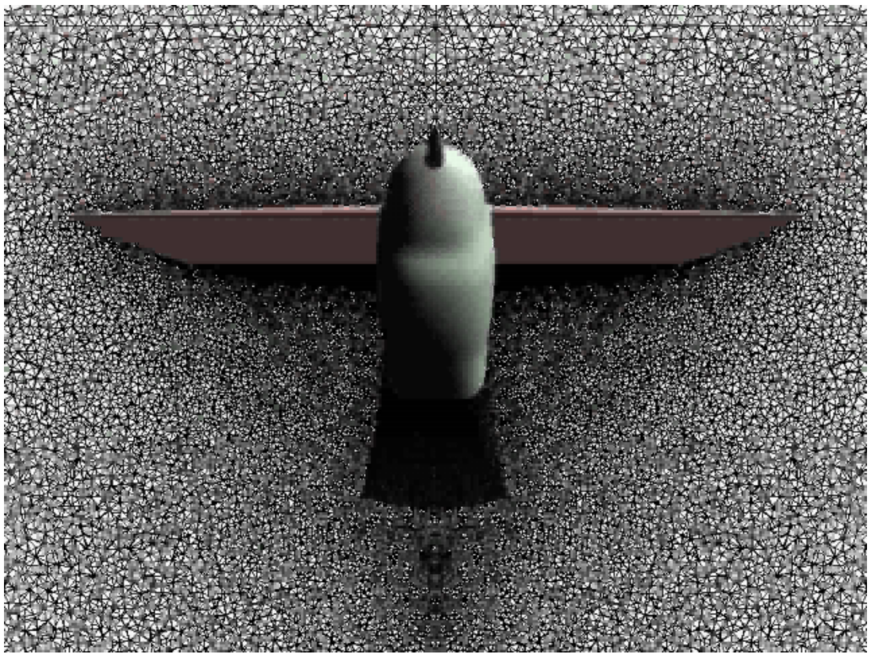
- Can read-from, or write-to, other processors “entities” whenever required

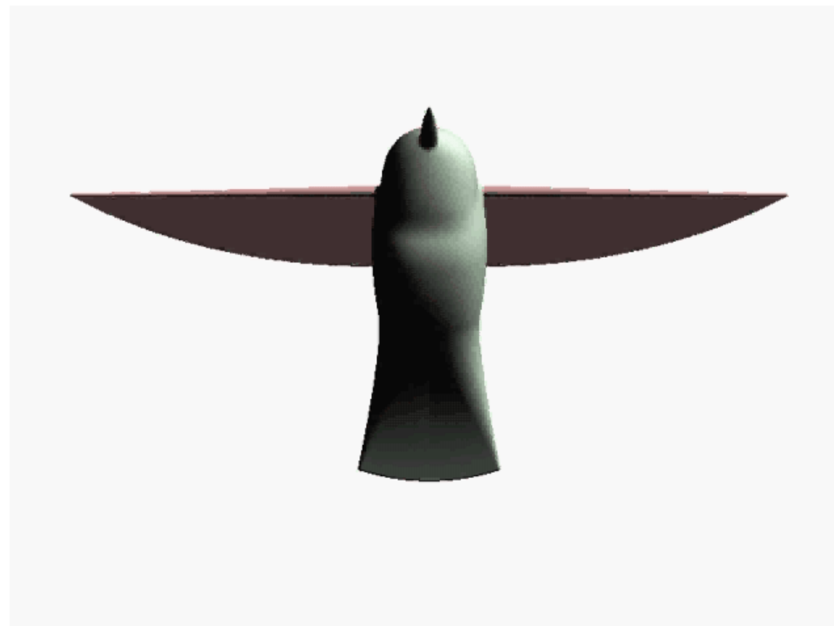
XFlow

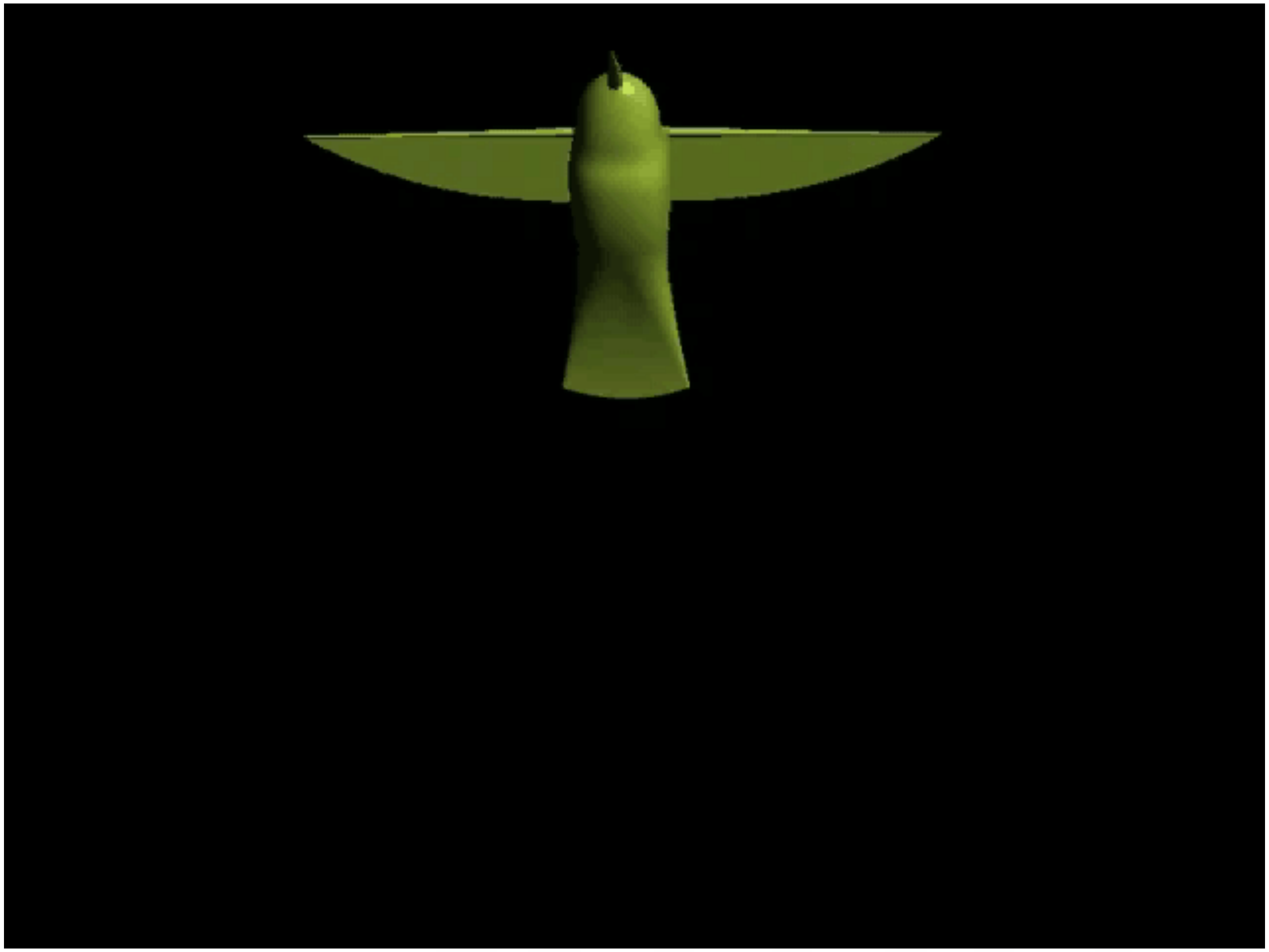
- What this method needs…
  - Each processor able to reference any arbitrary elements, faces, or nodes across entire mesh
  - Each processor able to modify any other processors portion of the mesh
  - Each processor able to search anywhere in the mesh
  - For performance, minimize these off-processor references by using smart mesh partitioning techniques
- Why MPI is not a good fit for this method
  - Can't arbitrarily read-from or write-to other processors data
  - Searches "stop" at processor/partition boundaries
    - Partition boundaries are "hard" (enforced) boundaries
  - A processor can't change and alter another processor's mesh structure
- Why is UPC good
  - You can do these kinds of things
    - Need to carefully use memory/process barriers



Dynamic Mesh







One-Sided Conclusions

- They offer improved productivity of explicit communication codes on all platforms
- They enable new algorithms that are difficult to impossible to code using two-sided communication
- They offer improved performance on Cray platforms